Efficient Algorithms for Listing k Disjoint st-Paths in Graphs

Roberto Grossi, Andrea Marino, Luca Versari

Università di Pisa, Italy, {grossi,marino,versari}@di.unipi.it

Abstract. Given a connected graph G of m edges and n vertices, we consider the basic problem of listing all the choices of k vertex-disjoint st-paths, for any two input vertices s, t of G and a positive integer k. Our algorithm takes $O(m)$ time per solution, using $O(m)$ space and requiring $O(F_k(G))$ setup time, where $F_k(G) = O(m \min\{k, n^{2/3} \log n, \sqrt{m} \log n\})$ is the cost of running a max-flow algorithm on G to compute a flow of size k. The proposed techniques are simple and apply to other related listing problems discussed in the paper.

1 Introduction

Listing paths of various kinds is a classical problem in graphs [7, 13, 17, 18], as it models problems in several contexts. The survey in [4], for instance, provides several bibliographical references to applications in biological sequence alignment, natural language processing, speech recognition, reconstruction of metabolic path-ways, gene regulation networks, motion tracking, message routing in communications networks, power line placement, vehicle and transportation routing, building evacuation planning, timing analysis of circuits, task scheduling, VLSI layout, communications and transportation network design.

In this paper we consider the basic problem of listing all choices of k disjoint st-paths for a given connected graph G of m edges and n vertices, where s, t are two vertices of G and k is a positive integer. In other words, we want to list all the possible ways of connecting s and t using k vertex-disjoint paths, for both directed and undirected graphs. We propose an algorithm that takes $O(m)$ time per listed solution, using $O(m)$ space and requiring $O(F_k(G))$ setup time, where $F_k(G) = O(m \min\{k, n^{2/3} \log n, \sqrt{m} \log n\})$ is the cost of running a max-flow algorithm on G to compute a flow of size k.

Our algorithm can be seen as an example where the textbook algorithms for computing strongly connected components (SCCs) and maximum flows are applied in a simple way. For this, we reduce the problem on k vertex-disjoint paths to the one on k *edge*-disjoint *trails*. A trail is a walk, possibly containing cycles, but with no repeated arc (see Section 2). We list the k edge-disjoint trails from s to t, considering just the trails for which there exist other (recursively checkable) $k-1$ edge-disjoint trails reaching t. The existence of the latter trails is guaranteed by the existence of a flow of size k. We exploit the fact that recomputing the flow for the next set of edge-disjoint trails takes $O(m)$ time instead of $O(F_k(G))$.

	Unbounded Length		Bounded Length	
	Target Given	Target not Given	Target Given	Target not Given
$k = 1$	[7] and this paper (directed): $O(m)$ [2] (undirected): optimal		[14]: $O(nm)$ (directed) and $O(m)$ (undirected) **this paper**: $O(m)$	
$k = 2$	**this paper**: $O(m)$	[1] and this paper: $O(m)$	[11]: hard	[15]: $O(nm)$ **this paper**: $O(m)$
$k \geq 3$	**this paper**: $O(m)$		[11]: hard	[15]: hard

Table 1. Cost per solution when listing the k vertex-disjoint (unbounded or bounded) paths originating from any given vertex s, whether the target t is given or not.

It is worth noting that some previous results, listed below, can be seen as variations of our problem. Our algorithm provides solutions within the same bounds as those reported in Table 1, and later discussed in Section 3, for the following ones:

- st-paths of a directed or undirected graph [7],
- shortest st-paths [8], a special case of bounded length st-paths [14],
- bubbles, i.e. pairs of vertex-disjoint directed st-paths, of fixed length [1],
- bubbles starting from a given vertex s [15],
- k-disjoint shortest paths with k given pairs of sources and targets [3].

Looking at the literature we also observe that the problem of finding k disjoint paths has been intensively studied. For $k = 1$, it is the classical problem of listing paths [2, 7, 13, 17]. For $k > 1$, it is related to graph connectivity (e.g. Menger's theorem [12] and its extensions) and the papers [1, 3, 8, 14, 15] in the above list present some variations. When k ordered pairs (s_i, t_i) of vertices are specified, and $s_i t_i$-paths are sought for, the problem of finding these k disjoint paths is NP-hard for arbitrary k in undirected graph [9], and even for $k = 2$ in directed graphs [5]. When k is fixed, an $O(n^3)$-time algorithm for undirected graphs can be obtained as a byproduct of the Robertson-Seymour papers on graph minors, and the bound has been recently improved to $O(n^2)$ time [10]. Note that the problem becomes polynomial when the k disjoint paths start from a source set $S = \{s_1, \ldots, s_k\}$ and reach a destination set $T = \{t_1, \ldots, t_k\}$, as the k disjoint paths can mix their sources and destinations (e.g. an $s_i t_j$-path with $j \neq i$ is allowed).[1] Still, listing the latter paths is interesting in applications.

Preliminaries. Given a directed graph $G = (V, E)$, we refer to its number of vertices as n and to its number of arcs as m. We refer to a walk as a sequence of adjacent vertices and arcs. In the following we distinguish between *trails* and *paths*: the former ones correspond to walks where the arcs are all distinct, while the latter ones are loopless trails, i.e. walks where both arcs and vertices are all distinct. Given $s, t \in V$, a st-trail, also denoted as $s \rightsquigarrow t$, (respectively st-path) is a trail (respectively a path) which starts in s and ends in t. Let τ be a $s \rightsquigarrow u$ trail, we denote as $G \setminus \tau$ as the graph G without the arcs in τ. For an arc (u, v), we denote as $s \rightsquigarrow u \cdot (u, v)$ the extension of τ with the new arc.

[1] Connect a dummy source s to each s_i, and each t_i to a dummy target t; then, run a max flow algorithm from s to t.

2 Edge-Disjoint Trails to a Single Target in Directed Graphs

Given a directed graph $G = (V, E)$ and some vertices $s_1, \ldots, s_k, t$, we first describe how to list all the sets of k edge-disjoint trails $\tau_1, \ldots, \tau_k$ such that τ_i is a $s_i t$-trail. After that, for vertices s, t, we will describe how to list edge-disjoint trails $\tau_1, \ldots, \tau_k$ such that τ_i is a st-trail.

Let us first focus on the well-known case $k = 1$. To generate all the $s_1 t$-trails, we can adopt a recursive algorithm that starts out with $u = s_1$ and considers the current trail $s_1 \rightsquigarrow u$ (seen as a sequence of arcs or a set of arcs with a little abuse of notation). It then explores in a recursive fashion, one at a time, each good neighbor v of u: namely, v is good iff v can reach t in the reduced graph $G' \equiv G \setminus (s_1 \rightsquigarrow u)$. For each good neighbor v, the recursion proceeds with the extended trail $s_1 \rightsquigarrow v \equiv (s_1 \rightsquigarrow u) \cdot (u, v)$.

For $k > 1$, we observe that not all choices of τ_1 are fruitful, as some of them could lead to dead ends for the remaining $k - 1$ trails. During the generation of each trail τ_1, we say that v is a *good neighbor* of u iff v reaches t in the reduced graph $G' \equiv G \setminus (s_1 \rightsquigarrow u)$, *and* there are also $k - 1$ disjoint trails from $s_2, \ldots, s_k$ to t: these two conditions can be equivalently seen as recursively checking the existence of k disjoint trails from $v, s_2, \ldots, s_k$ to t in G'.

We can therefore see the above generation of k disjoint trails as a recursive scheme, where the current $s_1 u$-trail ($s_1 \rightsquigarrow u$) has been already explored (initially, $u = s_1$). The main two cases are handled as follows.

- If $u = t$, the currently found trail $s_1 \rightsquigarrow u$ is τ_1. Moreover, if $k > 1$, recursively proceed with the $k - 1$ disjoint trails from vertices $s_2, \ldots, s_k$, setting $u := s_2$ and $G := G \setminus \tau_1$ (and noting that all these solutions will have this τ_1 fixed).
- If $u \neq t$, continue to generate the feasible trails τ_1 extending $s_1 \rightsquigarrow u$ (as prefix) in G. For this, extend u with its good neighbors: for each good neighbor v of u, proceed recursively with the extended trail $s_1 \rightsquigarrow v \equiv (s_1 \rightsquigarrow u) \cdot (u, v)$, traversing arc (u, v) and setting $u := v$.

The resulting recursion tree is illustrated in Figure 1(a). It is made up by a top tree for all the feasible choices of τ_1, where its leaves on the first dashed level are in one-to-one correspondence with these choices. Each such leaf is the root of the recursion tree for τ_2, where the leaves on the second dashed level are in one-to-one correspondence with these choices. Each such leaf is the root of the recursion tree for τ_3, and so on. In the specific case of Figure 1(a), nodes a, b, c, d, e indicate the possible choices for τ_1, while nodes f, g, h (respectively j, l, m) indicate the possible choices for τ_2 when $\tau_1 = b$ (respectively $\tau_1 = e$). Finally, nodes u, v, x, y correspond to k-sets of disjoint trails $\tau_1, \ldots, \tau_k$ where τ_i is the trail traversed at the i-th dashed line in the trail towards r in this tree.

In the following, we will call *nodes* the ones in the recursion tree to distinguish them from the *vertices* in the input graph.

Introducing the certificate. In order to make this recursion efficient within our claimed bounds, we introduce a certificate for each node in the recursion tree,

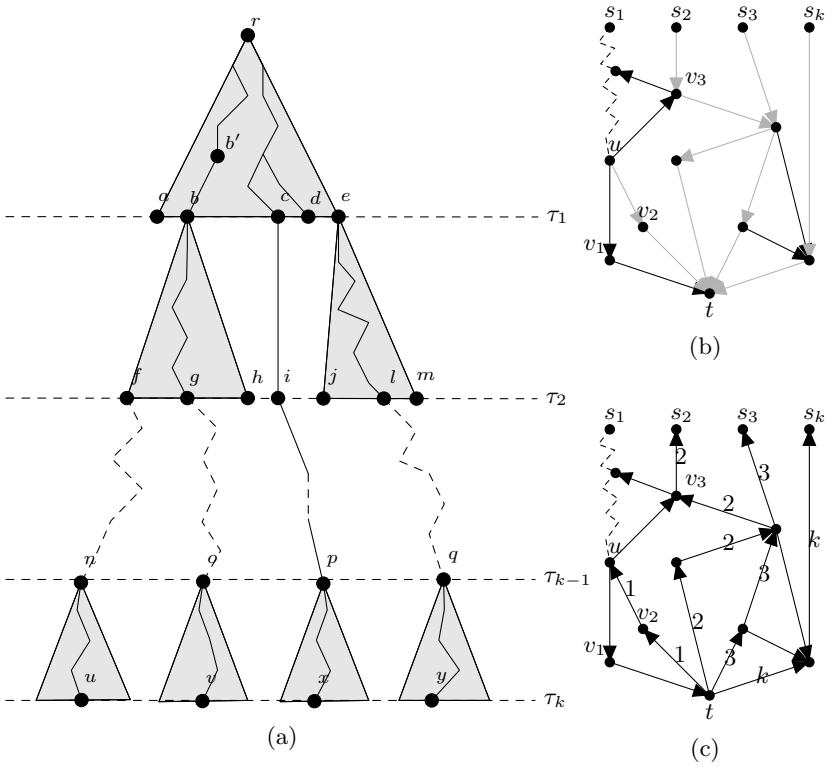

Fig. 1. (a) Recursion tree. (b) The paths $\pi_1, \ldots, \pi_k$ in gray with the dashed trail $s_1 \rightsquigarrow u$ and (c) the corresponding certificate C.

remarking the difference between trails and paths. We observe that the partial trail $s_1 \rightsquigarrow u$ uniquely identifies a node in the recursion tree where branching occurs by considering vertex u and its good neighbors. We want to keep track of a choice of k disjoint trails $\tau_1, \tau_2, \ldots, \tau_k$ from $s_1, \ldots, s_k$ to t, so that $s_1 \rightsquigarrow u$ is a prefix of τ_1. We observe that there exist k disjoint trails from $u, s_2, \ldots, s_k$ to t iff there exist k disjoint *paths* from $u, s_2, \ldots, s_k$ to t. Hence let $\pi_1, \pi_2, \ldots, \pi_k$ be the latter paths, and $G' \equiv G \setminus (s_1 \rightsquigarrow u)$ be the reduced graph, with the arcs in the partial trail removed.

Definition 1. *The* certificate *is an augmented graph $C = (V_C, E_C)$ defined in terms of the partial trail $s_1 \rightsquigarrow u$ and the paths $\pi_1, \pi_2, \ldots, \pi_k$, where the arcs belonging to the latter paths are reversed*[2] *and labeled to keep track of their membership to the paths, namely,*

- *$V_C = V$ is equal to the vertex set of G, and*

[2] Note that the certificate can be seen as the residual network in max-flow on arcs with 0-1 capacities.

- $E_C = \{(x, y) : (y, x) \in E(G') \cap \Pi(G)$ *or* $(x, y) \in E(G') \setminus \Pi(G)\}$, *where* $G' \equiv G \setminus (s_1 \rightsquigarrow u)$ *and* $\Pi(G) = \cup_{i=1}^{k} \pi_i$ *denotes the set of arcs belonging to the paths.*
- *Each arc* (x, y) *in* E_C *such that* $(y, x) \in \pi_i$ *(*$1 \leq i \leq k$*) is endowed with the label* i*.*

An example of certificate for the paths $\pi_1, \ldots, \pi_k$ and the trail $s \rightsquigarrow u$ in Figure 1(b) is shown in Figure 1(c), where each arc with label i belongs to a path π_i from s_i to t reversed.

Lemma 1. *Given a trail* $s_1 \rightsquigarrow u$*, the certificate* C *can be built in* $O(F_k(G))$ *time.*

Proof. Given $s_1, \ldots, s_k$ and t, we modify G by connecting $s_1, \ldots, s_k$ to a dummy vertex s. We can apply the Ford Fulkerson algorithm to get in $O(km)$ time k disjoint paths from s, and hence from $s_1, \ldots, s_k$, to t. In order to use a faster max-flow algorithm, like [6], we can do the following. Run the max-flow algorithm to find all the edges E' used by a flow from s to t of size k, then add k arcs from t to s to E'. In order to build our certificate, we need to find out k disjoint paths using the edges in E'. To this aim, we notice that all the vertices in the graph induced by the edges in E' have the in-degree equal to their out-degree. As this graph is an Eulerian multidigraph, we can find an Eulerian cycle involving s which by deleting the k arcs (t, s) splits into k disjoint st-trails. By deleting possible cycles in these trails, we get k disjoint st-paths. □

As discussed next, we employ the certificate to

- guarantee that there is at least one good neighbor of u, i.e, no dead ends for the current node in the recursion tree;
- locate the next good neighbor of u and save space (by avoiding to keep lists of good neighbors at each node in the recursion stack);
- quickly skip unary nodes in the recursion tree, i.e. when u has just a single good neighbor.

In this way the recursion tree in Figure 1(a) is actually flattened as a single tree where each node has at least two children. The benefit is clear by allocating a budget of $O(m)$ time on each child for every node, since we obtain an $O(m)$ cost per listed solution as a result. On the downside, since C extends the partial trail $s_1 \rightsquigarrow u$ to a specific choice of k disjoint trails, we have to characterize how to explore the other choices and, for each such choice, how to update C without computing it from scratch each time (hence, with a lower cost than that stated in Lemma 1).

At least one good neighbor of u*.* This part follows immediately from the definition of certificate, as the vertex v following u in the trail τ_1 can be characterized as the only vertex in G' that is neighbor of u and has a reverse arc in C. For this, we call v the *favorite* neighbor of u.

Next good neighbor of u. Apart from the favorite neighbor of u, mentioned above, we want to identify all the remaining good neighbors v of u. Also, the identification of the good neighbors should rely on a property that holds for any choice of the certificate C, so that the underlying C does not really matter to perform this task correctly, as shown below.

Lemma 2. *Given trail $s_1 \rightsquigarrow u$, for any certificate C and for any arc (u, v), we have that v is a good neighbor of u iff either v is the favorite neighbor of u in C or there is a cycle in C going through (u, v).*

Proof. If v is the favorite neighbor of u in C, we have nothing to prove. Otherwise, we want to prove that there is another certificate C' in which v is the favorite neighbor of u if and only if (u, v) is involved in a cycle in C. Consider the flow f that corresponds to certificate C and suppose that there is another flow d' on the same network that goes through (u, v). Then, it is well known that there will be a cycle in the residual network (i.e. in C) that contains the symmetric difference of the edges in the two flows. Moreover, the converse also holds. So we have another flow (and so another certificate) in which v follows u if and only if there is a cycle in C that uses (u, v). □

As an example, consider Figure 1(c). The good neighbors of u are v_2 (favorite neighbor) and v_1 (since it creates a cycle in C). On the other hand, v_3 is not a good neighbor for u as it does not satisfy both these conditions.

An immediate application of Lemma 2 to the nodes in the recursion tree has an excessive cost in terms of space and time.

Let us start with space. Rather than storing the list of good neighbors at each node in the recursive stack, which may take $\Omega(m)$ space since the same vertex can appear several times in the partial trail $s_1 \rightsquigarrow u$, we want to find the next good neighbor v' (if any) starting from the knowledge that the current good neighbor is v. Since vertices are numbered, it suffices to produce the list of good neighbors on the fly each time, and then select v' as the smallest one that is greater than v. This costs $O(m)$ time and fits the $O(m)$-per-children budget allocated to each node in the recursion tree.

Lemma 3. *Given trail $s_1 \rightsquigarrow u$ and any certificate C for it, the sorted list of good neighbors can be returned in $O(m)$ time and space.*

Proof. By Lemma 2, one neighbor is the favorite one that can be retrieved from C. As for the remaining ones, we compute the strongly connected components (SCCs) of C. Now, each arc (u, v) that is in a cycle must belong to a SCC that contains u. Thus scanning the SCCs identifies these arcs, as required by Lemma 2. Radix sorting these neighbors gives the wanted list. Total cost is $O(m)$ time and space. □

As for the time, rather than building the certificate in each node of the recursion tree as stated in Lemma 1, we prefer to compute the certificate from the parent or a child, taking $O(m)$ time instead of $O(F_k(G))$. In other words, we run the max-flow algorithms mentioned in Lemma 1 only once, to find one

certificate at the beginning of the recursion. After that, all the other certificates are computed using the method based on the following lemma.

Lemma 4. *Given the current node of the recursion tree and any certificate C for its trail $s_1 \rightsquigarrow u$, the certificate for the parent or for a child of the node can be computed in $O(m)$ time and space.*

Proof. Let $s_1 \rightsquigarrow u'$ be the trail in the parent node, where u is a good neighbor of u' and so $s_1 \rightsquigarrow u = (s_1 \rightsquigarrow u') \cdot (u', u)$. The new certificate is obtained from C by just adding the reversed arc (u, u') to C with label 1. Indeed, note that (u', u) was deleted in C, as C refers to $G \setminus s_1 \rightsquigarrow u$, and in the new certificate we are setting $\pi_1 := (u', u) \cdot \pi_1$.

We now show how the certificate modifies while going into a child node. Let $s_1 \rightsquigarrow v = (s_1 \rightsquigarrow u) \cdot (u, v)$ be the trail in the child node, where v is a good neighbor of u. Adding (u, v) may kill at most two paths in C: the path π_1 as we are possibly choosing another path replacing it, and a path π_i with $i \neq 1$, which could traverse (u, v). (Note that π_i is killed as we want disjoint paths, and no other π_j with $j \neq 1, i$ can traverse (u, v) for the same reason.) Observing that C can be seen as a residual network, we use the property of augmenting paths in Ford-Fulkerson algorithm. In particular, we have $k - 2$ flows (using a dummy source connected to $v, s_2, \ldots, s_k$), so we can run twice the $O(m)$-time algorithm to find an augmenting path to bring back the flow to k. This immediately translates into finding k disjoint trails.

Note that a final check is required to see if we still have paths (loop less trails) in the reversed arcs of the resulting certificate: in that case, we just remove the loops from the trails in $O(m)$ time. □

No unary nodes in the recursion tree. We need to avoid unary (i.e. single-child) nodes in the recursion tree. We have two kinds of situations to deal with as illustrated in Figure 1(a). A unary node can occur when producing a trail τ_i (as there is only one good neighbor), or when switching on a leaf from τ_i to the root that will generate τ_{i+1} (when for the given choice of $\tau_1, \ldots, \tau_i$, there is only one trail τ_{i+1} that exist). Looking at Figure 1(a), the former case corresponds to unary paths which are internal to a tree, as for instance the one from b' to b, while the second case corresponds to chains which remain unary across the trees, as for instance the one from c to p. In particular, this latter situation can give rise to a dependency on k in the time complexity ($O(mk)$) that suitably disappears by avoiding unary nodes.

We handle both kinds of situations using a simple "fast forward" technique that takes $O(m)$ time to skip maximal paths of unary nodes in the recursion tree, so that we actually obtain a compact recursion tree where each node always has two or more children.

Lemma 5. *Given the current node of the recursion tree and any certificate C for its trail $s_1 \rightsquigarrow u$, we can skip a unary chain starting from u in the recursion tree in $O(m)$ time and space.*

Proof. It is an extension of what discussed in the proof of Lemma 3. We compute the SCCs in the certificate C as before. We observe that if u is unary then the SCC containing u is trivial. Since the SCCs form a DAG, we take the sequence $S_1, \ldots, S_l$ of SCCs that are traversed by π_1 (see Definition 1 for the definition of π_i), where S_1 is the trivial SCC containing u and S_l is the SCC containing t. We take the first vertex u' along π_1 (towards t) that follows u and belongs to a non-trivial SCC S_j. Now we know that there are at least two good neighbors for u', which can be found in S_j as already discussed in the proof of Lemma 3. This handles the former situation.

To handle the latter situation, we observe that when we traverse $S_1, \ldots, S_l$ we always find trivial SCCs. So we switch to π_2, which we know from C, and take the sequence $S'_1, \ldots, S'_l$ of SCCs that are traversed by π_2. We stop in the first $S'_{j'}$ that is non-trivial. If it does not exist, we switch to π_3, and iterate the same approach. Either we stop at a non-trivial SCCs, and we proceed as in the proof of Lemma 3, or we find all trivial SCCs till π_k, so we obtain a solution to list.

In both situations, the cumulative cost is $O(m)$ time and space as we always traverse distinct SCCs. □

Theorem 1. *Given a directed graph G and its distinct vertices $s_1, \ldots, s_k, t$, all the k-sets of edge-disjoint trails $\tau_1, \ldots, \tau_k$, such that τ_i is a $s_i t$-trail for $1 \leq i \leq k$, can be listed with $O(m)$ time cost per solution. Initial setup time is $O(F_k(G))$ and space usage is $O(m)$.*

Proof. By definition of good neighbors, given a partial trail $s_i \rightsquigarrow u$ and the already generated trails $\tau_1, \ldots, \tau_{i-1}$, a simple induction shows that we generate all the k disjoint trails having $\tau_1, \ldots, \tau_{i-1}$ fixed and prefix $s_i \rightsquigarrow u$ for the ith trail. There is no solution listed twice for the same reason. Also, suppose by contradiction that there exists a solution that is not generated. Let $\tau_1, \ldots, \tau_{i-1}$ fixed and prefix $s_i \rightsquigarrow u$ be the node in the recursion tree that fails to generate it. By definition of good neighbor, this is a contradiction, as we identify the next v to be taken, so that $s_i \rightsquigarrow u$ can be extended to $s_i \rightsquigarrow u \cdot (u, v)$ instead, where v is the next vertex on the missed trail. Hence, all the trails are correctly listed once. Since we spend $O(m)$ time in each node of the recursion tree, and there are no unary nodes, we get $O(m)$ time per solution. The space is $O(m)$ as a trail can be so long, and we use constant memory per node in the recursion stack, which is of depth $O(m)$. □

We can now deal with the case where we have a single source s.

Theorem 2. *Given a directed graph G and two vertices $s, t \in V$, with $s \neq t$, all the k-sets of edge-disjoint trails $\tau_1, \ldots, \tau_k$, such that τ_i is a st-trail for $1 \leq i \leq k$, can be listed with $O(m)$ time cost per solution, setup time $O(F_k(G))$, and space usage $O(m)$.*

Proof. Let the *k-starting sets* be the k-subsets of s's neighbors, i.e. $\{s_1, \ldots, s_k\} \subseteq N(s)$, such that there are k disjoint paths $\pi_1, \ldots, \pi_k$ (and hence trails) where

π_i is an $s_i t$ path. If we apply Theorem 1 to every k-starting set, we obtain our claim.

Since we cannot explore all the possible subsets of $N(s)$, we again use a recursive approach to generate just the k-starting sets of s. Suppose that $N(s)$ is sorted by vertex numbering. At each recursive step we divide the vertices of $N(s)$ into three groups:

- I = those vertices that will be part of any starting set generated from the current recursive call;
- H = those vertices that won't be in any starting set generated from the current recursive call;
- U = those vertices that may or may not be in a starting set.

Initially, both I and H are empty, and $U = N(s)$.

During the recursion, we keep the invariant that U forms a contiguous suffix of the sorted sequence $N(s)$. At each step, we compute a flow f of cardinality k starting from $\{s\} \cup I$ on the graph obtained from G by erasing all incoming edges in s and all outgoing edges from s to any vertex in $I \cup H$. More precisely, we require one unit of flow to leave each vertex in I, none to leave any vertex in H, and $k - |I|$ to leave s.[3] We observe that we are actually enforcing a flow of cardinality k that uses first all the vertices in I. This flow is computed from scratch once, at the beginning of the recursion. In the rest of the recursion, it is updated through the calls using the same ideas as in the proof of Lemma 4.

We compute the residual network R_f of f, and the SCCs of R_f. We can thus build the set W of vertices $v \in U$ such that (s, v) is involved in a cycle in R_f.

If W is empty, then there is exactly one k-starting set that satisfies the constraints given by the sets I and H, so we can output I (plus any other neighbor of s involved in the f) as k-starting set, and return from the recursive call.

Otherwise, let v be the smallest vertex in W. Since (s, v) is involved in a cycle in R_f (as any other vertex in W), there must exists at least two flows, both traversing the vertices I and avoiding those in H, such that one flow traverses (s, v) and the other does not traverse (s, v). This means that the current recursive call generates two further calls: the former where v goes into I, and the latter v goes into H. In both calls, all vertices in U that precede v go into either I or H, according to their role in f: a vertex $x \in U$ with $x < v$, goes to I if x is traversed by the flow for the call at hand, or goes to H otherwise.

Hence, each calls returns at least one solution, and each internal call gives raise to two further calls. Hence the cost per solution is bounded by the cost of a single call, which is $O(m)$. Space cost is $O(m)$ with setup time $O(F_k(G))$.

It is worth observing that for each generated k-starting set, we can produce in $O(m)$ time the initial certificate needed (with the empty path as the current path). In this way we do not pay each time the setup cost $O(F_k(G))$ when applying Theorem 1 to the k-starting set. □

[3] This flow can be achieved by creating a dummy vertex s' connected to all the vertices in I with capacity 1 and to s with an arc of capacity $k - |I|$.

3 Applications to Related Problems

In this section, we show how to use the results in Section 2 to solve a variety of related problems. We firstly discuss some easy variations, including listing vertex disjoint paths, and then we focus on the problem of listing bounded-length fixed-source k-disjoint-paths with $k = 2$, since for $k \geq 3$ (as summarized in Table 1) the latter problem is hard.

3.1 Vertex-disjoint paths and other variations

Undirected graphs and vertex disjoint paths. Since we are interested in edge disjoint trails, in the case of undirected graphs, it is not sufficient to consider two opposite arcs in place of an undirected edge, as we want to avoid to traverse one arc in the solution if the other has been used. To this aim, well-known reductions [16] allow to easily extend our Theorem 2 to undirected graphs. Moreover, by transforming each vertex v into an arc (v_{in}, v_{out}), putting arcs (w_{out}, v_{in}) and (v_{out}, z_{in}) for each in-neighbor w and each out-neighbor z of v, vertex disjoint paths reduce to edge disjoint trails: as each of the k edge disjoint trails composing a solution do not use the same arc twice, we get the following.

Theorem 3. *Given a directed or undirected graph G and two vertices $s, t \in V$, with $s \neq t$, all the k-sets of vertex disjoint paths $\pi_1, \ldots, \pi_k$, such that π_i is a st-path for $1 \leq i \leq k$, can be listed with $O(m)$ time cost per solution, setup time $O(F_k(G))$, and space usage $O(m)$.*

It is worth observing, that Theorem 3 generalizes the result in [1] for any k, getting the same bounds for $k = 2$, as $O(F_2(G))$ is $O(m)$.

Cycles involving two vertices. Given an undirected graph, Theorem 3 easily extends to enumerating all the simple cycles that contain two given vertices s, t: it is enough to enumerate all the pairs of vertex-disjoint paths that connect s to t. As for directed graphs, we recall that finding even one directed cycle involving two vertices is NP-hard [5].

Multiple Sources vs Multiple Targets. Further variations can be considered. For instance, listing all the k-sets of edge disjoint trails (or vertex disjoint paths) from any subset of $x \geq k$ sources $\{s_1, \ldots, s_x\}$ to any subset of $y \geq k$ targets $\{t_1, \ldots, t_y\}$. This can be easily solved in the same bounds claimed by Theorem 2 and Theorem 3, by simply attaching $s_1, \ldots, s_x$ to a dummy source s and $t_1, \ldots, t_y$ to a dummy target t. We remark that this problem is different from the one considered by Robertson and Seymour and Kawarabayashi et al. [10] concerning the so called *disjoint paths problem*, since in their case, as said in the introduction, the sources and targets are paired and any algorithm is forced to find a k-set of vertex disjoint paths respecting this pairing.

Fixed source vs Variable Target. Let us now consider the following: given a directed graph $G = (V, E)$ and $s \in V$, list all the k-sets of edge disjoint st-trails (or vertex disjoint st-paths) for any $t \in V$. In order to solve this problem, we need to compute for which t there is at least a solution. By applying the results in Theorem 2 and Theorem 3, we get a total time cost of $O(nF_k(G) + \alpha m)$ with $O(m)$ space, where α is the number of solutions. In undirected graphs, for $k = 1, 2, 3$ the total time cost becomes simply $O(\alpha m)$ as the n flow computations can be replaced by the computation of k-connected components.

3.2 Bounded-length fixed-source two-disjoint-paths

In this section we discuss the following problem which deals with st-paths of bounded lengths.

Problem 1. Let $G = (V, E)$ be a directed graph, $s \in V$ and $\ell \in \mathbb{N}$. List all the k-sets of vertex disjoint st-paths of length at most ℓ, simultaneously for any $t \in V$.

State-of-art results are summarized in the last two columns of Table 1. If target t is given a priori along with source s, the only case which can be solved with output-sensitive bounds is for $k = 1$. Hence we focus on the case $k = 2$ with t not given a priori (Problem 1), showing how to apply similar ideas to Section 2. As this will make use of a subroutine to list all the st-paths of length at most ℓ, we will also focus on the case $k = 1$ where both s and t are given a priori.

Given s, we will make use of ball $B_\ell(G)$, which is defined as the graph induced by the vertices at distance at most ℓ from s in G. For the sake of simplicity, we assume wlog that s has zero indegree in G. Similarly to Section 2, we build first a st-path π_1 for some t and then the second vertex disjoint st-path π_2. Once the st-path π_1 is fixed, the suitable choices for π_2 are all the st-paths of length at most ℓ in G where vertices in π_1 except s and t have been deleted.

Let us now focus on the building process for π_1. Recall that while building π_1 we have to guarantee that π_1 can be completed in a solution, i.e. there is at least a suitable π_2 which can be paired with π_1. Let u be the current vertex explored during the recursion (at the beginning $u = s$) and let π'_1 be the su-path of length h with $h \leq \ell$ built until that point. As in the previous section we have to explore all the good neighbors of u. To discover them, we will employ an auxiliary graph G', obtained from G by removing all the vertices in π'_1 except s. A neighbor v of u is good whether there is a way to complete the partial solution, i.e. v is in G' and, moreover, in G' there is a vertex t such that there is a vt-path π''_1 with length at most $k-h$ and there is also a st-path π_2 of length at most ℓ, where π''_1 and π_2 are vertex disjoint. For each good neighbor v of u we recur by deleting v from G'. Note that this guarantees that we generate just simple paths and, by definition of good neighbors, the path π_2 does not overlap with π'_1 and π''_1.

Good Neighbors and Certificate. Similarly to the previous section, we maintain a certificate in order to recognize the good neighbors v for u. It may happen that u itself may be reached in at most ℓ steps from s without using any nodes used in π_1. In this case, we know that u is a valid target and so we proceed with generating all possible π_2. Moreover, it is also possible that π_1' is a prefix of a valid path (this must be true if u is not a valid target, as we want to avoid dead-ends). In this case the certificate C is simply a ut-path from u to some vertex t in $B_\ell(G')$, such that t is the only vertex of the path in $B_\ell(G')$. Let v be the neighbor of u in C, where v corresponds to the *favorite* neighbor defined in Section 2. The other good neighbors are the vertices in G' that are in $N(u)$ and can reach at least a vertex in $B_\ell(G')$ using at most $\ell - h$ arcs. These can be easily computed in $O(m)$ by collapsing the vertices in $B_\ell(G')$ into a single vertex b and then running a backward BFS from b in G' truncated at distance $\ell - h$. Let R be the vertices reached by this BFS: the good neighbors for u are all the vertices in $N(u) \cap R$ and a certificate for each of them is their path to b. Hence, for any good neighbor, we have a child node in the recursion tree and for each of them we can build the certificate in $O(m)$. When backtracking from a child to its parent we can rebuild the certificate in the same way in $O(m)$ thus avoiding to store the past certificates in the recursion stack as in Section 2.

Fast Forward. We now show how to guarantee that in each recursion node we always have at least *two* children, we spend $O(m)$ time in each recursion node. We want to skip unary chains in the recursion tree in $O(m)$ time as in Lemma 5. Let u be the current vertex, let π_1' be the su-path of length h with $h \leq \ell$ built until that point, and let π_1'' be the ut-path (for some t) in the current certificate C. There is a unary chain for instance if the neighbor v of u in π_1'' is its only good neighbor or if the path in π_1'' is the only feasible completion for π_1', that is for each vertex in π_1'' except t, which is in $B_\ell(G')$, there is only one good neighbor. Precisely, we want to find the first vertex w in the path π_1'' that has at least two neighbors in $O(m)$ in order to skip all the intermediate vertices and continue the recursion from that one. This task can be easily addressed by modifying the certificate update as shown next.

1. Scan the vertices in the path π_1'' in their order checking whether there are at least two good neighbors as follows: let b be the vertex obtained by collapsing the vertices in $B_\ell(G')$, for each vertex w in the path π_1'' check whether there is an arc (w, z) not in π_1'' such that z is at distance at most $\ell - h$ from b.
2. Let w be the first vertex satisfying this condition. We directly add all the edges on the uw-path in π_1'' to π_1' going directly to the recursion node corresponding to w.

With respect to Section 2, we are ignoring fast-forwarding across the recursion trees for the different values of k, as since $k = 2$, linear dependencies on k do not afflict here our asymptotic time costs per solutions.

Generating π_2. Once a st-path π_1 has been generated, we have to generate all the st-paths of length at most ℓ in the graph G' which is obtained by deleting

from G all the vertices in π_1 except s and t. We show next that this can be done with a slight modification of the process for generating π_1 shown above. As in the above case, when recurring at u and enlarging the su-path π'_2 with length h, the certificate C is a path towards t of length at most $\ell - h$. Each recursive node explores all the good neighbors of u, i.e. the ones leading to t with at most $\ell - h$ arcs. The certificate update and the computation of the good neighbors can be done exactly as in the above case by simply replacing $B_\ell(G)$, and hence b, with the given target t. Hence, in order to understand which are the good neighbors, it suffices to do a backward BFS from t truncated at distance $\ell - h$. The good neighbors are the ones in $N(u)$ reached by this BFS and the certificate for them is their path to t. By replacing $B_\ell(G)$ with t in the π_1 generation, we get also fast-forward which allows to skip unary chains in the recursion tree in $O(m)$ time: as before, in $O(m)$ time we can scan the vertices w of the su-path in C looking for arcs (w, z) with z at distance at most $\ell - h$ from t.

Since both the trees generating π_1 and π_2 have no unary nodes and each internal node costs $O(m)$ time, we get a $O(m)$ cost per solution. Concerning space, we observe that certificates, as in Section 2, are updated not only while going from a parent node to a child, but also while backtracking. For this reason, the recursion stack for both π_1 and π_2 trees is just made by good neighbors lists whose cardinalities sum up to $O(m)$ since each vertex appears just once in each root to leaf path.

As a result, we get the following, which holds also for undirected graphs by simply replacing each edge with a pair of opposite arcs.

Theorem 4. *Let $G = (V, E)$ be a directed or undirected graph and $\ell \in \mathbb{N}$ and consider Problem 1.*

- *Given $s, t \in V$, there is an algorithm which lists all the st-paths of length at most ℓ with $O(m)$ time costs per solution (case $k = 1$).*
- *Given $s \in V$, there is an algorithm which lists all pairs of disjoint st-paths of length at most ℓ for any $t \in V$ with $O(m)$ time costs per solution (case $k = 2$).*

Setup time and space usage are $O(m)$ in both cases.

As a final remark, Theorem 4 also extends to the problem of listing all the cycles with bounded length involving a given vertex.

Bibliography

[1] E. Birmelé, P. Crescenzi, R. A. Ferreira, R. Grossi, V. Lacroix, A. Marino, N. Pisanti, G. A. T. Sacomoto, and M. Sagot. Efficient bubble enumeration in directed graphs. In *Proc. SPIRE 2012*, pages 118–129, 2012.

[2] E. Birmelé, R. A. Ferreira, R. Grossi, A. Marino, N. Pisanti, R. Rizzi, and G. Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In *Proc. SODA 2013*, pages 1884–1896, 2013.

[3] T. Eilam-Tzoreff. The disjoint shortest paths problem. *Discrete Applied Mathematics*, 85(2):113–138, 1998.

[4] D. Eppstein. k-best enumeration. In *Encyclopedia of Algorithms*, pages 1003–1006. 2016.

[5] S. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, Feb. 1980.

[6] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM (JACM)*, 45(5):783–797, 1998.

[7] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975.

[8] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.

[9] R. M. Karp. On the computational complexity of combinatorial problems. *Networks*, 5(1):45–68, Jan. 1975.

[10] K.-i. Kawarabayashi, Y. Kobayashi, and B. Reed. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B*, 102(2):424–435, 2012.

[11] C.-L. Li, S. T. McCormick, and D. Simchi-Levi. The complexity of finding two disjoint paths with min-max objective function. *Discrete Applied Mathematics*, 26(1):105–115, 1990.

[12] K. Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

[13] R. C. Read and R. E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

[14] R. Rizzi, G. Sacomoto, and M. Sagot. Efficiently listing bounded length st-paths. In *Proc.IWOCA 2014*, pages 318–329, 2014.

[15] G. Sacomoto, V. Lacroix, and M.-F. Sagot. A polynomial delay algorithm for the enumeration of bubbles with length constraints in directed graphs and its application to the detection of alternative splicing in rna-seq data. In *Proc. WABI*, pages 99–111. Springer, 2013.

[16] A. Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer Science & Business Media, 2003.

[17] R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM J. on Comput.*, 2(3):211–216, 1973.

[18] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications ACM*, 13:722–726, 1970.